



Buenas prácticas de programación en Python

Lección 1: Desarrollo de código guiado por pruebas.

ÍNDICE

1.	Presentación y objetivos.....	2
2.	Test unitarios.....	3
3.	Test unitarios con pytest	5
	Test unitarios para varias funciones.....	9
4.	Test unitarios con unittest	14
5.	Limitaciones test unitarios.....	19

Desarrollo de código guiado por pruebas.

1. PRESENTACIÓN Y OBJETIVOS

En esta lección aprenderemos qué es un test unitario, porqué utilizarlos en nuestras implementaciones y cuáles son sus principales ventajas e inconvenientes. Por otro lado, aprenderemos a utilizar dos de las herramientas más comunes en Python para crear test unitarios. En concreto desarrollaremos un conjunto de tests para validar las diferentes funciones implementadas en nuestros programas.



Objetivos

- Comprender la importancia del desarrollo de código guiado por pruebas.
- Comprender cuáles son las principales limitaciones de los test unitarios.
- Conocer y utilizar la herramienta *pytest* para validar nuestro código.
- Conocer y utilizar la herramienta *unittest* para validar nuestro código.
- Conocer las principales diferencias y limitaciones entre *pytest* y *unittest*.

2. TEST UNITARIOS

En programación, realizar un **test unitario** es el proceso por el cual podemos comprobar el correcto funcionamiento de un fragmento de la totalidad del código. En términos generales, la idea consiste en escribir un conjunto de casos de prueba para cada función o método implementado para asegurar el correcto funcionamiento de cada módulo por separado.



Importante

El principal objetivo es asegurar que cada unidad funciona correctamente y que además, el código que ha sido implementado cumple con su cometido.

Cuando implementamos un conjunto de pruebas unitarias sobre nuestro software, hemos de comprobar que se cumplan los siguientes requisitos:

- **Automatizable.** Una vez implementado el conjunto de test unitarios, estos deberían tener la capacidad de ser ejecutados y validados por sí mismos, sin necesidad de la intervención manual del programador.
- **Completas.** Dichos tests deben cubrir la mayor cantidad posible del código desarrollado.
- **Reutilizables.** Han de ser ejecutables más de una vez.
- **Independientes.** El funcionamiento de un test debe ser independiente del resto de pruebas unitarias.

- **Profesionales.** Deben ser desarrollados minuciosamente y con la misma profesionalidad con que ha sido desarrollado el resto del código.



Presta atención

Probar que el código desarrollado funciona correctamente antes de que éste realice la tarea para la que ha sido creado implica una amplia variedad de beneficios.

Sin embargo, cabe destacar que las pruebas unitarias no podrán descubrir todos los errores del código ya que no es una tarea trivial la de anticipar todos los casos especiales de entradas que puede recibir cualquier función bajo test.

Las pruebas unitarias solo son efectivas si se usan en combinación con otras pruebas de software

Además del inconveniente mencionado anteriormente, es necesario señalar que escribir y mantener pruebas es un trabajo duro, por lo que utilizar herramientas de terceros que nos faciliten esta tarea es una gran idea.

3. TEST UNITARIOS CON PYTEST

Como sabemos, **Python cuenta con** una gran cantidad de librerías que pueden ser utilizadas para resolver problemas de distinta índole. Es por ello que, también cuenta entre sus repositorios de algunas **librerías** diseñadas para facilitar la tarea del desarrollador a **crear test unitarios**. En este tema vamos a estudiar el uso de **pytest** como herramienta para implementar test unitarios en nuestros programas.



Importante

Utilizar test unitarios en tus desarrollos aumenta la confianza para con tus implementaciones, ya que se tiene la certeza de que el código se comporta como espera y asegura que los cambios en el código no causarán un daño mayor.

Con **pytest**, las tareas comunes requieren de menos código mientras que las tareas más avanzadas pueden ser realizadas con una amplia variedad de comandos y complementos que incluye **pytest**, ahorrándose de este modo una cantidad significativa de tiempo de trabajo.

Entrando a la práctica, debemos instalar en primer lugar el paquete **pytest**. Para ello, accedemos a una terminal y lo instalamos por medio del gestor de paquetes **pip**. (Nótese que en este manual se está haciendo uso del sistema operativo **Ubuntu**, por lo que la forma de instalar un paquete se hace mediante *`pip install <nombre_paquete>`* .

Consulte sus apuntes de la asignatura Fundamentos de Python para ver cómo utilizar el gestor de paquetes *pip* con otro Sistema Operativo, como Windows).

```

ramon@ramon-rd: ~/Escritorio/ev
(fp) (base) ramon@ramon-rd:~/Escritorio/ev$ pip install pytest
Collecting pytest
  Downloading pytest-6.2.3-py3-none-any.whl (280 kB)
    | 280 kB 2.1 MB/s
Requirement already satisfied: pluggy<1.0.0a1,>=0.12 in /home/ramon/miniconda3/lib/python3.8/site-packages (from pytest) (0.13.1)
Collecting iniconfig
  Using cached iniconfig-1.1.1-py2.py3-none-any.whl (5.0 kB)
Requirement already satisfied: packaging in /home/ramon/miniconda3/lib/python3.8/site-packages (from pytest) (20.9)
Collecting py>=1.8.2
  Using cached py-1.10.0-py2.py3-none-any.whl (97 kB)
Requirement already satisfied: toml in /home/ramon/.local/lib/python3.8/site-packages (from pytest) (0.10.2)
Requirement already satisfied: attrs>=19.2.0 in /home/ramon/miniconda3/lib/python3.8/site-packages (from pytest) (20.3.0)
Requirement already satisfied: pyparsing>=2.0.2 in /home/ramon/miniconda3/lib/python3.8/site-packages (from packaging->pytest) (2.4.7)
Installing collected packages: iniconfig, py, pytest
Successfully installed iniconfig-1.1.1 py-1.10.0 pytest-6.2.3
(fp) (base) ramon@ramon-rd:~/Escritorio/ev$

```

Para comprobar que se ha instalado correctamente podemos ejecutar el comando *py.test -h*, y nos mostrará la ayuda de dicho paquete:

```

ramon@ramon-rd: ~/Escritorio/ev
(fp) (base) ramon@ramon-rd:~/Escritorio/ev$ py.test -h
usage: py.test [options] [file_or_dir] [file_or_dir] [...]


positional arguments:
  file_or_dir

general:
  -k EXPRESSION          only run tests which match the given substring
                        expression. An expression is a python evaluable
                        expression where all names are substring-matched against
                        test names and their parent classes. Example: -k
                        'test_method or test_other' matches all test functions
                        and classes whose name contains 'test_method' or
                        'test_other', while -k 'not test_method' matches those
                        that don't contain 'test_method' in their names. -k 'not
                        test_method and not test_other' will eliminate the
                        matches. Additionally keywords are matched to classes
                        and functions containing extra names in their
                        'extra_keyword_matches' set, as well as functions which
                        have names assigned directly to them. The matching is
                        case-insensitive.
  -m MARKEXPR           only run tests matching given mark expression.
                        For example: -m 'mark1 and not mark2'.
  --markers              show markers (builtin, plugin and per-project ones).
  -x, --exitfirst        exit instantly on first error or failed test.
  --fixtures, --funcargs show available fixtures, sorted by plugin appearance
                        (fixtures with leading '_' are only shown with '-v')
  --fixtures-per-test    show fixtures per test
  --pdb                 start the interactive Python debugger on errors or
                        KeyboardInterrupt.
  --pdbcls=modulename:classname
                        start a custom interactive Python debugger on errors.
                        For example:
                        --pdbcls=IPython.terminal.interactiveshell:TerminalPdb

```

Una vez instalado, procedemos a realizar un ejemplo guiado para explicar de una manera sencilla cómo funciona la herramienta *pytest*.

En primer lugar, creamos un nuevo fichero python con el nombre 'test_prueba1.py'. En este programa crearemos un test unitario para comprobar si la suma de dos números enteros se realiza o no correctamente.



Presta atención

Nótese que aquellos ficheros python que contengan el código relacionado con los test unitarios, deben comenzar con la palabra *test*.

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
GNU nano 4.8 test_prueba1.py
import pytest

def test_suma_enteros():
    x = 3
    y = 2
    resultado = 6
    assert x+y == resultado
```

Para ejecutar el test ejecutamos `pytest <nombre_test>` o `py.test`:


```

ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ pytest test_prueba1.py
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/ramon/Escritorio/BPP/Leccion2
collected 1 item

test_prueba1.py F [100%]

===== FAILURES =====
_____ test_suma_enteros _____

    def test_suma_enteros():
        x = 3
        y = 2
        resultado = 6
        assert x+y == resultado
E       assert (3 + 2) == 6

test_prueba1.py:7: AssertionError
===== short test summary info =====
FAILED test_prueba1.py::test_suma_enteros - assert (3 + 2) == 6
===== 1 failed in 0.05s =====
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$

```

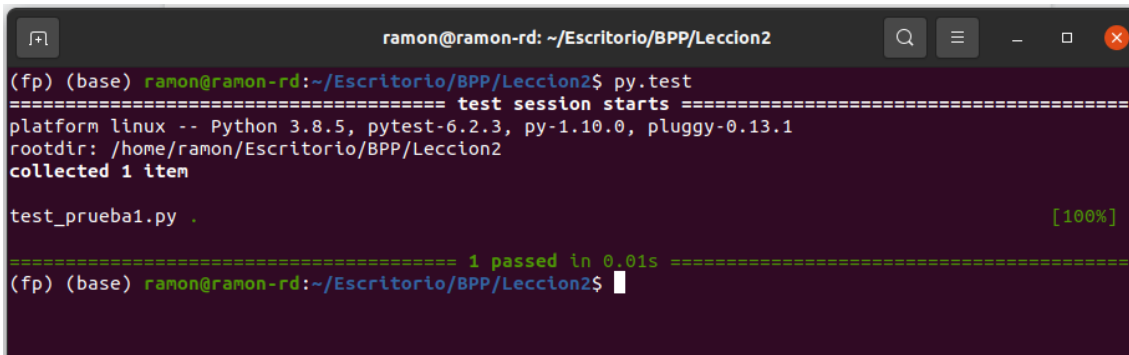
Como se puede comprobar, el test ha fallado ya que la operación realizada para sumar dos enteros no ha sido correcta. En concreto, le hemos indicado al test que compruebe si la suma de los enteros 3 y 2 es 6 y como era de esperar, el test ha fallado. Si modificamos el código para que la suma de 3 y 2 sea 5, comprobaremos que ahora todo está en orden.

```

GNU nano 4.8 test_prueba1.py
import pytest

def test_suma_enteros():
    x = 3
    y = 2
    resultado = 5
    assert x+y == resultado

```

A terminal window titled 'ramon@ramon-rd: ~/Escritorio/BPP/Leccion2' showing the execution of 'py.test'. The output indicates a successful test session with 1 item collected and 1 test passed in 0.01s. The test file is 'test_prueba1.py'.

```
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ py.test
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/ramon/Escritorio/BPP/Leccion2
collected 1 item

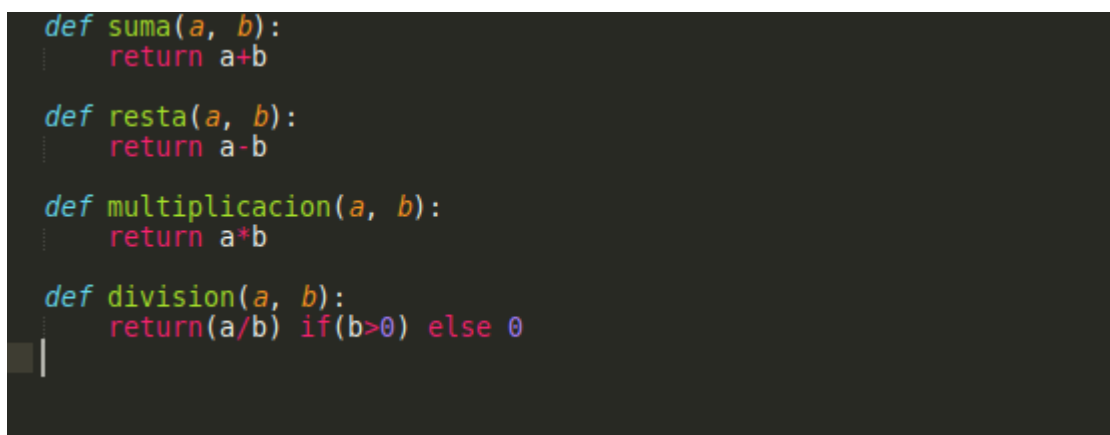
test_prueba1.py . [100%]

===== 1 passed in 0.01s =====
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$
```

Test unitarios para varias funciones

En la sección anterior hemos estudiado cómo implementar un test unitario para verificar el correcto funcionamiento de un fragmento de código. Imaginad ahora que nuestro programa principal se llama 'operaciones.py' en el que hemos definido un conjunto de operaciones matemáticas para trabajar con números enteros. Siguiendo las instrucciones del apartado anterior, podemos desarrollar uno o varios test unitarios para verificar el correcto funcionamiento de las diferentes funcionalidades de nuestro programa.

Veamos un ejemplo: en la siguiente captura de pantalla se muestra el código principal de nuestro programa. En este hemos definido cuatro funciones, cada una de ellas resuelven las operaciones de suma, resta, multiplicación y división respectivamente.

A code editor showing four Python functions: suma, resta, multiplicacion, and division. Each function takes two arguments, a and b, and returns the result of the operation. The division function includes a conditional check to avoid division by zero.

```
def suma(a, b):
    return a+b

def resta(a, b):
    return a-b

def multiplicacion(a, b):
    return a*b

def division(a, b):
    return(a/b) if(b>0) else 0
```

A continuación, podemos definir un test unitario asociado a cada una de las funciones anteriores de la siguiente manera:

```
import pytest
import operaciones

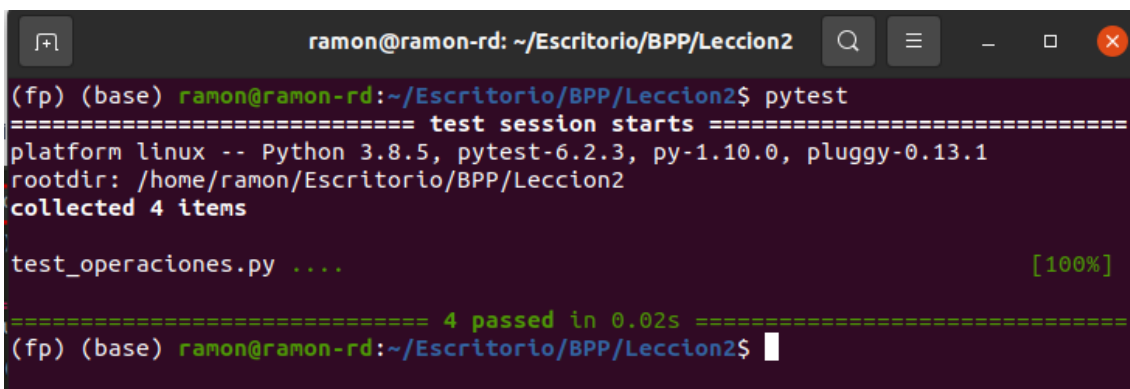
def test_suma():
    x = 3
    y = 2
    resultado = 5
    assert resultado == operaciones.suma(x,y)

def test_resta():
    x = 3
    y = 2
    resultado = 1
    assert resultado == operaciones.resta(x,y)

def test_multiplicacion():
    x = 3
    y = 2
    resultado = 6
    assert resultado == operaciones.multiplicacion(x,y)

def test_division():
    x = 6
    y = 2
    resultado = 3
    assert resultado == operaciones.division(x,y)
```

Nótese que hemos importado el fichero operaciones, que contiene las operaciones de suma, resta, multiplicación y división comentadas anteriormente. Una vez definido el test unitario podemos verificar su funcionamiento:



```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ pytest
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/ramon/Escritorio/BPP/Leccion2
collected 4 items

test_operaciones.py .... [100%]

===== 4 passed in 0.02s =====
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$
```

Ahora imagine que queremos añadir la operación potencia (elevar un número):

```
def suma(a, b):  
    return a+b  
  
def resta(a, b):  
    return a-b  
  
def multiplicacion(a, b):  
    return a*b  
  
def division(a, b):  
    return(a/b) if(b>0) else 0  
  
def elevarNumero(base, exponente):  
    return base*exponente
```

NOTA: hemos introducido el error intencionadamente. En la función señalada no estamos calculando la potencia de un número, sino que estamos realizando una simple multiplicación.

Tenga en cuenta el test unitario definido para comprobar el funcionamiento de la función de elevar:

```
def test_elevar():  
    base = 2  
    exponente = 8  
    resultado = 256  
    assert resultado == operaciones.elevarNumero(base,exponente)
```

Puesto que tenemos la certeza de que $2^8=256$, si al ejecutar el test nos arroja un error tendremos la certeza de que éste proviene de la propia implementación de calcular la potencia:

```

ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
platform linux -- Python 3.8.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/ramon/Escritorio/BPP/Leccion2
collected 5 items

test_operaciones.py ....F [100%]

===== FAILURES =====
test_elevar

def test_elevar():
    base = 2
    exponente = 8
    resultado = 256
    assert resultado == operaciones.elevarNumero(base,exponente)
E   assert 256 == 16
E   + where 16 = <function elevarNumero at 0x7f371ddc8af0>(2, 8)
E   +   where <function elevarNumero at 0x7f371ddc8af0> = operaciones.elevarN
umero

test_operaciones.py:32: AssertionError
===== short test summary info =====
FAILED test_operaciones.py::test_elevar - assert 256 == 16
===== 1 failed, 4 passed in 0.06s =====
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$

```

Arreglemos el código y volvamos a ejecutar el test:

```

def suma(a, b):
    return a+b

def resta(a, b):
    return a-b

def multiplicacion(a, b):
    return a*b

def division(a, b):
    return(a/b) if(b>0) else 0

def elevarNumero(base, exponente):
    return base**exponente

```

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ pytest
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/ramon/Escritorio/BPP/Leccion2
collected 5 items

test_operaciones.py ..... [100%]

===== 5 passed in 0.01s =====
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$
```

4. TEST UNITARIOS CON UNITTEST

Unittest es una alternativa a pytest para implementar test unitarios en Python. Esta herramienta es una de las más utilizadas en Python y permite realizar pruebas desde el propio código, heredando de la clase unittest.

Con unittest las pruebas pueden arrojar uno de los siguientes resultados:

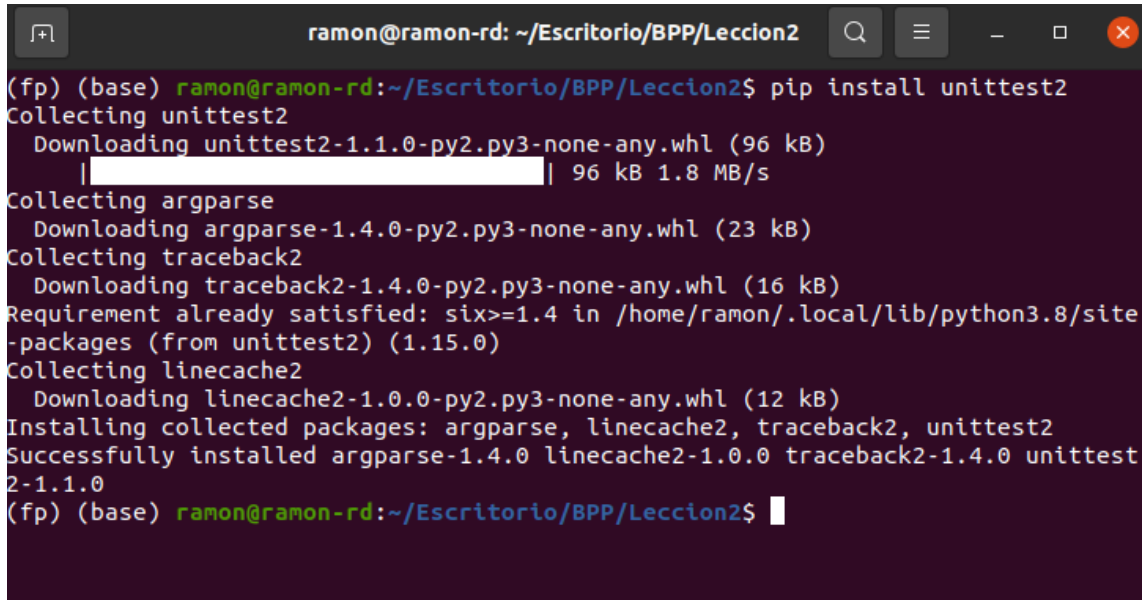
- **OK.** Nos indica que la prueba ha sido ejecutada correctamente y que no se ha encontrado ningún problema.
- **FAIL.** Indica que la prueba no ha podido ser ejecutada y sería necesario incluir una excepción (try/except).
- **ERROR.** Indica que el fragmento de código no ha pasado el test.



Importante

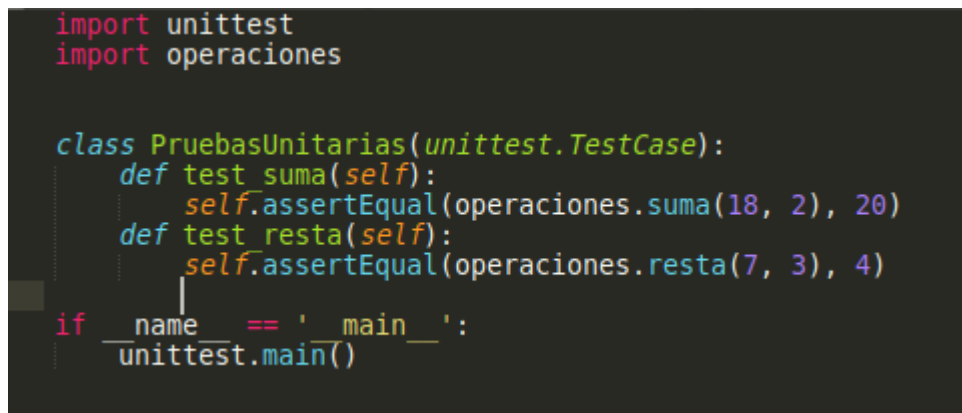
Implementar test unitarios con la herramienta unittest puede ser más laboriosa que con pytest. Aunque en este manual mostramos el funcionamiento de cada una de ellas, es decisión del estudiante elegir la que más se adapte a sus necesidades.

Para poder utilizarlo necesitamos instalar el paquete unittest2:



```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ pip install unittest2
Collecting unittest2
  Downloading unittest2-1.1.0-py2.py3-none-any.whl (96 kB)
    |████████████████████| 96 kB 1.8 MB/s
Collecting argparse
  Downloading argparse-1.4.0-py2.py3-none-any.whl (23 kB)
Collecting traceback2
  Downloading traceback2-1.4.0-py2.py3-none-any.whl (16 kB)
Requirement already satisfied: six>=1.4 in /home/ramon/.local/lib/python3.8/site-packages (from unittest2) (1.15.0)
Collecting linecache2
  Downloading linecache2-1.0.0-py2.py3-none-any.whl (12 kB)
Installing collected packages: argparse, linecache2, traceback2, unittest2
Successfully installed argparse-1.4.0 linecache2-1.0.0 traceback2-1.4.0 unittest2-1.1.0
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$
```

Una vez instalado, podemos utilizarlo como se muestra en el siguiente ejemplo:



```
import unittest
import operaciones

class PruebasUnitarias(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(operaciones.suma(18, 2), 20)
    def test_resta(self):
        self.assertEqual(operaciones.resta(7, 3), 4)

if __name__ == '__main__':
    unittest.main()
```

Tras ejecutarlo en un terminal (lo hacemos del mismo modo que ejecutamos cualquier otro programa python) obtenemos el siguiente resultado:


```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ python test_unittest.py
..
-----
Ran 2 tests in 0.000s

OK
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$
```

En el fragmento de código anterior podemos identificar principalmente que a diferencia de la herramienta pytest, con unittest necesitamos:

- Implementar una clase que herede de la clase `unittest.TestCase`
- Necesitamos utilizar la palabra reservada *assertEqual* para comprobar que el resultado de nuestra función implementada devuelve el resultado correcto.



Importante

Si deseamos utilizar unittest, debemos memorizar (o al menos tener accesible) el conjunto de palabras reservadas creadas para realizar las diferentes comprobaciones en los test unitarios.

En la documentación oficial de unittest podemos encontrar una tabla con los métodos que son utilizados con mayor frecuencia:

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

Otros métodos que pueden resultar de interés:

Method	Checks that
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>
<code>assertGreater(a, b)</code>	<code>a > b</code>
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>
<code>assertLess(a, b)</code>	<code>a < b</code>
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>
<code>assertRegex(s, r)</code>	<code>r.search(s)</code>
<code>assertNotRegex(s, r)</code>	<code>not r.search(s)</code>
<code>assertCountEqual(a, b)</code>	<code>a</code> and <code>b</code> have the same elements in the same number, regardless of their order.

Method	Used to compare
<code>assertMultiLineEqual(a, b)</code>	strings
<code>assertSequenceEqual(a, b)</code>	sequences
<code>assertListEqual(a, b)</code>	lists
<code>assertTupleEqual(a, b)</code>	tuples
<code>assertSetEqual(a, b)</code>	sets or frozensets
<code>assertDictEqual(a, b)</code>	dicts

5. LIMITACIONES TEST UNITARIOS

En esta sección mostraremos algunas de las limitaciones de los test unitarios, para que el estudiante comprenda a un mayor nivel de detalle cómo funcionan dichos test y cuándo estos no pueden ser utilizados. Debido a su facilidad de uso, en este caso práctico haré uso de la herramienta **pytest**.

Imagine que hemos implementado un programa que simula el funcionamiento de una máquina tragaperras. El programa debe generar de forma aleatoria tres valores que pueden ser: cereza, manzana, naranja, kiwi y melón. Si los cinco valores generados coinciden (es decir, si se genera todo cerezas, todo manzanas, todo naranjas o todo melones) habremos ganado el premio. ¿Cómo podemos implementar un test unitario que compruebe el correcto funcionamiento de este método? Nótese que el test unitario podría comprobar que la salida de nuestra función son caracteres de texto (el conjunto de caracteres que conforman la palabra manzana, cereza, etc.), que los valores son iguales o diferentes, pero no podemos hacer otra cosa.

A continuación mostramos el fragmento de código descrito anteriormente y un ejemplo de su ejecución:

```
import numpy as np

def checkList(lst):
    elem = lst[0]
    resultado = True
    for i in range(1, len(lst)):
        if(lst[i] != elem):
            resultado = False

    return resultado

def tragaperras():
    opciones=["Cereza", "Manzana", "Naranja", "Kiwi", "Melón"]
    tirada = np.random.randint(0, len(opciones), size=len(opciones))
    print("El resultado de su tirada es: ")
    [print(opciones[i]) for i in tirada]
    resultado = checkList(tirada)
    print("Ha ganado el premio") if(resultado) else print("Mala suerte, inténtelo de nuevo")

tragaperras()
```

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion2
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$ python tragaperras.py
El resultado de su tirada es:
Kiwi
Melón
Kiwi
Manzana
Naranja
Mala suerte, inténtelo de nuevo
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion2$
```

¿Cómo implementaría un test asociado a este código?

Nótese que escribir un test unitario sobre un fragmento de código cuyo resultado está directamente relacionado con números aleatorios no puede realizarse debido al alto número de combinaciones posibles.

Sin embargo, sí podríamos construir un test unitario asociado a la función *checkList* y a otras funciones auxiliares (si las hubiera) que no estén ligadas a un resultado aleatorio. Ante este tipo de casuísticas es el desarrollador quien debe asegurarse que la implementación es correcta y haber realizado todas las pruebas necesarias para validar su código.

PUNTOS CLAVE

En esta lección hemos aprendido:

- | Qué son los test unitarios y porque es importante incluirlos en nuestros desarrollos.
- | Utilizar la herramienta **pytest** para validar los diferentes métodos implementados en nuestro código.
- | Utilizar la herramienta **unittest** para validar los diferentes métodos implementados en nuestro código.

